An Approach to the Generation of High-Assurance Java Card Applets

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
Ph. +1-650-493-6871 Fax +1-650-424-1807
http://www.kestrel.edu/~coglio
coglio@kestrel.edu

1 Introduction

1.1 Smart Cards

A *smart card* [11], also known as *integrated circuit card* or *chip card*, consists of an electronic chip embedded into a plastic substrate. The size of the plastic substrate is the same as a credit card or smaller. So, the chip is very small and hence has relatively limited processing and memory capabilities, even though these capabilities keep increasing thanks to the continual advancements in semiconductor technology. A smart card communicates with the external world via metal contacts on the surface of the card or via an antenna wound into the card. The signals exchanged through the contacts or the antenna encode commands to and responses from the card. The card's memory and functionality can be only accessed via these commands and responses. So, smart cards are easily portable, cost-effective computing devices that can *securely* store and process information.

For example, a smart card can store a private key and use it to sign or encrypt/decrypt data. Since the card only recognizes and processes certain commands (e.g., to sign data provided in the command, returning the signature as a response), there is no way to extract the value of the private key as a response from the card. The value of the private key could be returned only if such a functionality had been explicitly programmed into the card (which hopefully has not). Smart cards also have built-in defenses to make it difficult (i.e., not cost-effective) to break their security via special hardware equipment. Current applications of smart cards include authentication, financial transactions, telephony, and the list is growing.

While smart cards *can* provide secure storage and processing of information, such a goal is fulfilled only if the smart card's software and hardware are *correct* with respect to the desired security policies. Correctness is of paramount

importance for smart cards: it is the main criterion for determining the quality of a smart card product.

1.2 Java Card

Java Card [2] is a version of Java for smart cards. From the programmer's point of view, a program in Java Card is written in a subset of Java (features such as floating point arithmetic and multi-dimensional arrays are not part of the subset) and makes use of different libraries than the usual Java libraries. Java Card libraries provide functionality to receive commands and send responses according to certain standardized protocols, to perform cryptographic computations, etc. In addition to these standard libraries, various industry-specific libraries are being defined (e.g., for telephony and banking). An application in Java Card is called a Java Card *applet*. Aside from the name, Java Card applets have very little to do with the Java applets embedded in Web pages.

A Java Card program's source code is compiled via a regular Java compiler, resulting in a number of class files (one for each class declared in the program), as usual. Next, a *converter* is run on these class files. The converter checks that the class files conform to the Java Card subset of Java, and in that case it produces a number of *CAP* (Converted APplet) files (one for each package constituting the program). The CAP files are optionally verified by a *verifier* and then loaded into the *Java Card Runtime Environment* (*JCRE*), which is a virtual machine running on the smart card. The JCRE sits on top of the smart card operating system, which in turn sits on top of the hardware. The JCRE includes an interpreter for bytecodes as well as an implementation of the Java Card libraries (mostly in native code). The CAP file format is optimized for loading into and execution by the JCRE.

The term *Java Card Virtual Machine* (*JCVM*) is an overloaded one. It is used to denote the interpreter of the JCRE, but it is also used to denote the ensemble of converter, verifier, and JCRE. The Java Virtual Machine (JVM) reads class files and executes them. Analogously, the JCVM, according to its second meaning just described, reads class files and eventually executes them, with conversion and verification taking place in the process. But while the converter and the verifier run outside the card, the JCRE runs on the card. In fact, this architecture is referred to as *split* virtual machine architecture. Given the limited memory and processing capabilities of smart cards, currently it is problematic to put converter and verifier on the card: this is the reason for the split architecture.

Java Card brings the benefits of the Java technology to the world of smart cards. There exist several smart card platforms, and until a few years ago it was difficult to develop portable smart card applications or to update the card contents after the issuance of the card. Also, the type safety of Java provides a good foundation for security. So, Java Card provides a uniform and secure platform for the development of smart card applications.

Other emerging standard platforms for smart cards are the MULTOS operating system [9] and Windows for Smart Cards [15].

2 High Assurance and Kestrel's Work

In order for a Java Card applet to be secure (according to the desired security policies), the applet must be correctly implemented. In addition, the platform on which the applet runs, i.e., the JCRE, must be correctly implemented. Since the JCRE sits on top of a smart card operating system and hardware, the operating system and the hardware must be also correctly implemented. Finally, all the tools used "between" the source code and the JCRE (namely compiler, verifier, and converter) must be correctly implemented.

In other words, the assurance of a Java Card applet depends on several, mutually dependent critical elements. Of course, high assurance is not a black-or-white property, but rather it is "measured" in shades of gray. Improving the correctness of any of the above critical elements contributes significantly to increasing the overall assurance of the applet.

Kestrel has been developing tools and techniques for the development of high-assurance software. Our Specware tool [8] provides capabilities to write formal specs, refine them, compose specs and refinements, generate running code from refined specs, and call mechanical theorem provers to prove the correctness of specs and refinements as well as putative properties of specs for validation purposes. Our Planware tool [1] is a domain-specific extension of Specware: it is a fully automatic generator of schedulers from high-level specs that are written by the user in a simple and intuitive tabular form. Planware internally translates these domain-specific specs into Specware specs and makes use of the Specware machinery to refine them to extremely efficient code. Other generators are currently being built at Kestrel (e.g., a generator of C code from Stateflow [7] diagrams).

Currently, we have projects underway to apply and extend our synthesis technology to Java Card. More precisely, we are working on the following two tasks:

1. synthesize implementations of the JCRE and related tools;
2. build an automatic generator of Java Card applets.

For the first task, we are building a formal spec of the JCRE in Specware. This spec will then be refined into a simulator first, then into a JCRE implementation that runs on some smart card. The formal spec includes the CAP file format, the interpretation of bytecodes, and the Java Card libraries. Our formal spec can serve as an unambiguous, precise version of the informal spec provided by Sun [13, 14, 12]. The simulator is a development tool that simulates the JCRE on a desktop computer, useful for testing and debugging purposes. Its correctness is obviously very important, because the developer expects it to faithfully represent the behavior of an actual card. We may also synthesize a converter and an off-card verifier. We are approaching this long-term task via vertical slices.

We have been previously successful at the specification and synthesis of certain security-critical components of the JVM. Since Java security is based on

Figure 1: User's view of the generator

type safety, we have studied the components of the JVM that are responsible for enforcing type safety. We have realized a complete mathematical spec of the Java bytecode verifier, which has been written in Specware and refined to a running implementation (see [6] for a preliminary account). In this process, we have designed various improvements over Sun's spec and implementation of bytecode verification [3], in particular the treatment of subroutines [4] and subtype checks [10]. We have also identified some bugs in Sun's spec and implementation, and proposed corrections [5]. Furthermore, we have developed a formal spec of the JVM class loading mechanisms along with their interplay with bytecode verification, and proved a type safety theorem [10] about our formalization.

The second task, namely the construction of a Java Card applet generator, is the main topic of this paper and is described in the next section.

3 An Architecture for an Applet Generator

This section describes work in progress that is in early stages of its development. So, many technical details have not been pinned down yet. Nevertheless, the description provides a high-level view of the approach that is being taken in this project.

3.1 User's View

From the user's point of view, the generator appears as shown in Figure 1. It is a box that takes an applet spec as input and automatically produces code implementing the specified applet. The spec is written in a domain-specific language that has a precise, mathematically defined semantics, while being at the same time easy and intuitive to use by smart card developers. The vocabulary of this domain-specific language includes smart cards concepts such as commands, reponses, keys, PINs, etc. Obviously, the constructs of this language are higher-level than the Java Card code that achieves the same functionality.

In other words, from the user's point of view the generator is a smart compiler that translates high-level specs written in a domain-specific language to lower-level Java Card code.

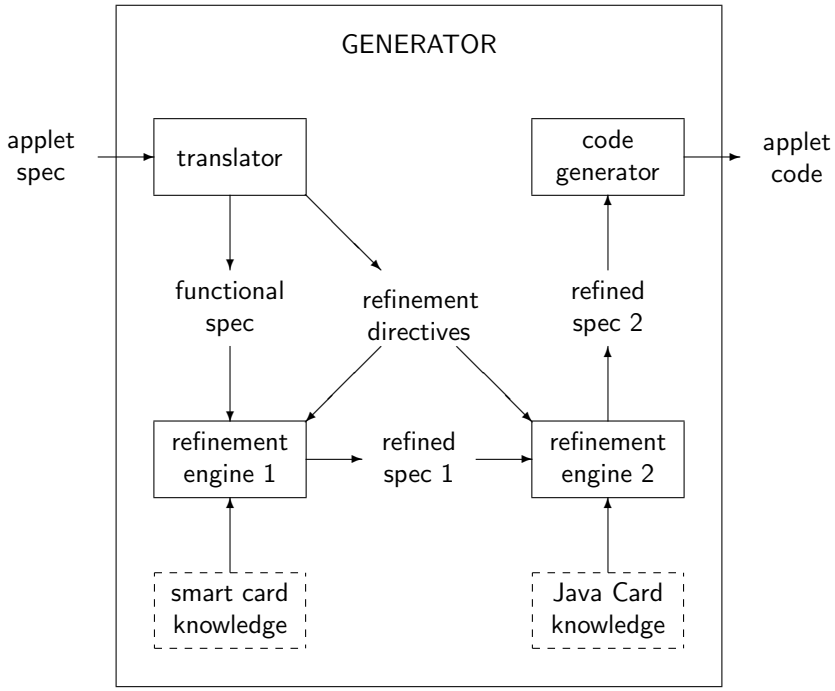

Figure 2: Inside the generator

3.2 Inside the Generator

Looking inside the generator, the structure appears as in Figure 2.

The first block inside the generator is a *translator* that produces a functional spec for the applet, written in the spec language of Specware. The spec language of Specware is a version of higher-order logic, so it is mathematically precise; however, the user is not exposed to it, because Specware specs are only produced (and processed, as described below) internally.

In addition to the functional spec, the translator also produces some *refinement directives*, i.e., information to guide the refinement of the functional spec. The nature of these refinement directives is best explained through the following example. Consider an applet that recognizes and processes five different commands. In the spec for this applet (the one written in the domain-specific language) the user declares symbols to denote these five commands, possibly accompanied by parameters, and uses these symbols to specify the processing of each of these commands. In the actual applet, these commands have to be encoded as byte sequences that are exchanged between the smart card and the external world, according to certain standardized protocols. So, the applet spec must include mappings of the symbolic commands to byte sequences. The functional spec produced by the translator expresses the high-level functionality of

the applet, and therefore only specifies commands symbolically. The encoding of commands as byte sequences is in fact a refinement of the functional spec, so the translator extracts from the applet spec this information as a refinement directive that is later used during the refinement of the functional spec.

The *first refinement engine* makes use of the Specware machinery to apply some refinements to the functional spec. The engine makes use of knowledge about smart cards, which is part of the generator. This is generic smart card knowledge, not specific to Java Card. For example, it includes information about the standardized protocols for smart card communication. This knowledge is embodied as Specware specs and refinements, which are suitably composed with and applied to the functional spec by the refinement engine, based on the refinement directives. The result of this process is a refined spec that is not specific to Java Card.

The *second refinement engine* also makes use of the Specware machinery to apply further refinements to the refined spec. It makes use of knowledge that is specific to Java Card, e.g., the API of the libraries for handling the communication protocols. This knowledge is also embodied as Specware specs and refinements. These are suitably composed with and applied to the refined spec, based on the refinement directives, to produce a more refined spec that is now specific to Java Card.

The spec produced by the second refinement engine is ready to be translated to Java Card by the *code generator*. The result is Java Card code that implements the applet spec provided by the user.

The reason for having two separate refinement steps, one independent from and the other dependent on Java Card, is to pave the way for future extensions of the generator where the same applet spec can be refined to different languages/platforms. For example, a third refinement engine could be added that is specific to MULTOS, including knowledge of MULTOS in the form of Specware specs and refinements. Another code generator could be added that transforms specs refined through this third refinement engine into C code (the language used in development for MULTOS). This MULTOS refinement engine and C code generator are "parallel" to the Java Card refinement engine and code generator: the user chooses which one to use. But the platform-independent refinement engine is used in both cases.

The main advantages of this architecture are:

1. the high assurance deriving from the use of Specware's mathematical foundations;

2. the modularity of the generator, which eases the evolution of the generator (e.g., improving the Java Card code generator to produce code with a smaller footprint, adding smart card knowledge about inter-applet communication, adding a new target platform).

3.3 Independent Certification

A generator greatly reduces human coding errors, especially when the generator is based on solid foundations such as Specware's. However, the question arises about the correctness of the generator itself. There may be bugs in the generator that affect the correctness of the generation process.

A similar concern applies to verification, and not only to synthesis. Even if a program has been verified via a theorem prover, model checker, or other tool, bugs in the verification tool may invalidate the correctness result of the program.

In verification, an approach to increase assurance is to have the verification tool produce an explicit proof (provided that the tool possesses this functionality) that can be checked with a simple proof checker. The proof checker is much simpler and smaller than the verification tool, and therefore more likely to be correct. After the proof has been checked, the correctness of the proof only depends on the correctness of the checker, and not of the tool used to produce the proof.

The Specware refinement process can produce proofs of each refinement step. The proofs for the various refinement steps can be composed together to obtain a proof of the overall refinement process. Proofs for the individual steps are produced by the transformations that are applied to the specs: since the transformations are (designed to be) correctness-preserving, there is a proof associated to each application of a transformation.

This means that the applet generator could be extended to generate, besides the applet code, also a proof that the applet code correctly implements the user's spec. The proof can then be checked independently, as shown in Figure 3. The *checker* takes as inputs the applet spec, the applet code, and the proof; it returns a yes/no answer, depending on whether the proof is valid with respect to the given applet spec and code. If the checker returns a positive answer, then the correctness of the code with respect to the spec only depends on the correctness of the smaller and simpler checker, and not any more on the correctness of the larger and more complex generator.

The synthesis approach constructs the correctness proof during the synthesis process, by composing smaller proofs corresponding to the various refinement steps. In contrast, an after-the-fact verification approach must construct the whole proof from the code only. The search space is therefore larger. In other words, the synthesis approach can provide more scalability.

4 Future Work

Since, as previously mentioned, this project is in its early stage, much of what has been described in the previous section is in fact future work. The work plan is to start with a simple generator, with relatively limited capabilities, and then add more and more capabilities to it. From the user's point of view, the evolution of the generator corresponds to an evolution of the domain-specific

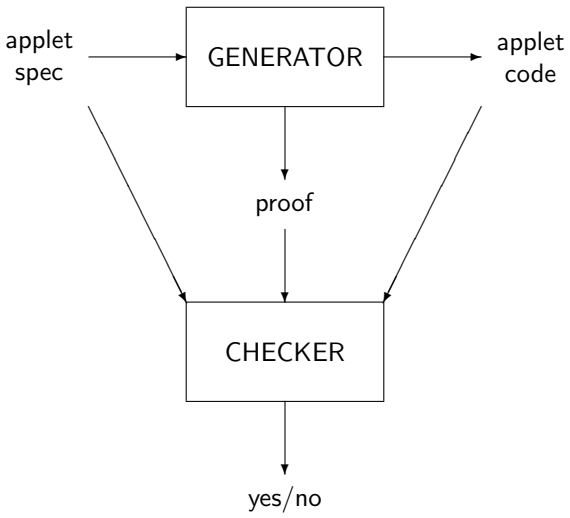

Figure 3: Independent certification

language and to an expansion of the set of applets that can be generated by the generator. Internally, the various components of the generator (including the knowledge used by the refinement engines) are evolved modularly.

While the first versions of the generator will target the Java Card platform, successive versions may target additional platforms, such as C/MULTOS.

References

[1] Lee Blaine, Li-Mei Gilham, Junbo Liu, Doug Smith, and Stephen Westfold. Planware: Domain-specific synthesis of high-performance schedulers. In *Proc. 13th Conference on Automated Software Engineering (ASE'98)*, pages 270–280, October 1998.

[2] Zhiqun Chen. *Java Card™ Technology for Smart Cards.* Addison-Wesley, 2000.

[3] Alessandro Coglio. Improving the official specification of Java bytecode verification. In *Proc. 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.

[4] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. Technical report, Kestrel Institute, December 2001.

[5] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, November 2001.

[6] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[7] The Math Works Inc. Stateflow user's guide, 2000.

[8] Kestrel Institute. Specware™. http://www.specware.org.

[9] MULTOS™. http://www.multos.com.

[10] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000. Long version available at http://www.kestrel.edu/java.

[11] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley and Sons, second edition, 2000.

[12] Sun Microsystems. Java Card™ 2.1.1 Application Programming Interface, May 2000. Available at `http://java.sun.com/javacard`.

[13] Sun Microsystems. Java Card™ 2.1.1 Runtime Environment (JCRE) Specification, May 2000. Available at `http://java.sun.com/javacard`.

[14] Sun Microsystems. Java Card™ 2.1.1 Virtual Machine Specification, May 2000. Available at `http://java.sun.com/javacard`.

[15] Windows for Smart Cards. http://www.microsoft.com/windowsce/smartcard.